Paxos STM – User Guide

Release 1.0

**Tadeusz Kobus, Maciej Kokociński,
Paweł T. Wojciechowski
www.it-soa.pl/jpaxos**

January 2011

Contents

Contents:

CHAPTER 1

Overview

Paxos STM is a robust implementation of an object-oriented Distributed Software Transactional Memory (DSTM) in Java, which is tolerant to machine crashes. It provides a convenient way of developing highly available and efficient services, by replicating a service on several machines and using atomic transactions to maintain replica consistency. Paxos STM has been inspired by two concepts: Software Transactional Memory (STM) [ST95] [BHG87] and Distributed Shared Memory (DSM), restricted to whole objects.

1.1 Main features

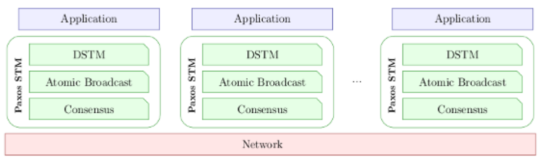
Data (objects) that are critical to the service are copied on network nodes running service replicas. Paxos STM ensures that reads and writes to these data on a given node, are consistently propagated to all replicas. In case of transaction conflicts, transactions are rolled back and restarted. Transaction atomicity and isolation are guaranteed.

Non-blocking transaction certification is viewed as a request to the distributed agreement protocol implemented by JPaxos. Following JPaxos's system assumptions and guarantees, Paxos STM fully supports the crash-recovery failure model. This is a key feature of our design.

Paxos STM employs a user friendly, error-resistant API that makes transactional objects and object replication transparent. It shifts the burden of tracking transactional object accesses to the framework, thus making programming easier and less error-prone. However, no language extensions have been made.

1.2 System architecture

Paxos STM is built in modular fashion with each building block providing recovery capabilities:



At the bottom of the stack of protocols used by Paxos STM is the JPaxos [JPaxosWeb] library which is responsible for solving distributed consensus. It provides fault-tolerance and is able to work in a network where messages can be lost. On top of JPaxos there is the Atomic Broadcast [M06] module which guarantees that all the nodes (also referred to as replicas) will eventually receive the same set of messages in the same order. This delivery property is necessary for the employed in the Paxos STM optimistic transaction certification algorithms [C09]. In the optimitic approach every transaction is executed in a way that is invisible to all the other concurrently executed transaction. When an attempt to commit the transaction is undertaken all the necessary information about the transaction (e.g. the sets containing objects read and written in the transaction, logical time representing the moment of the start of the transaction) is broadcast using atomic broadcast mechanism to all the replicas. Therefore each node can independently and deterministically perform the transaction certification so all the replicas eventually decide whether the transaction can be committed or it has to be rolled back and restarted. Employing this certification scheme results in no additional synchronization between the nodes during the execution of the transactions. In order to enhance the level of fault tolerance and to reduce the latency in accessing the shared objects, all the shared objects are replicated on each node.

A complete description of Paxos STM appeared in the technical reports [KKW09] [KKW10].

CHAPTER 2

Requirements

Paxos STM requires the following components to be installed on the target system:

- execution platform: Java Runtime Environment 1.6
- compilation platform: Java Software Development Kit 1.6
- additional libraries:
  - JPaxos (GPL v3)
  - Javassist (MPL, LGPL)
  - Objenesis (Apache Licence v2.0)

CHAPTER 3

Compilation and configuration

3.1 Configuration

Paxos STM relays on JPaxos library; configuration of Paxos STM is largely determined by the configuration of JPaxos. Therefore the first thing that has be done to run an application employing Paxos STM is to set up is JPaxos parameters (`paxos.properties` file in the projects main directory). For details of JPaxos configuration please check JPaxos' documentation. However, the most important parameters are:

network configuration

IP addresses and ports on which JPaxos will run on each replica, e.g.:

```
process.0 = 192.168.1.100:2021:3001
process.1 = 192.168.1.101:2022:3002
process.2 = 192.168.1.102:2023:3003
```

This entry defines three replicas identified by numbers 0, 1 and 2. These numbers are used during execution of an application utilizing Paxos STM (see Running an application employing Paxos STM for details). Each instance of JPaxos requires two available ports.

failure model configuration

There are a number of different failure models Paxos STM can work with. Some are not yet available due to the undergoing development of JPaxos library, as noted below):

- **crash-stop** - the process which crashes ceases communication with other processes and never recovers; calling the commit primitive does nothing; very fast but does not provide any guarantees concerning fault tolerance
- **full stable storage** - on each node each message passed by JPaxos is logged to the stable storage so upon crash and recovery all the missed messages will be delivered to the recovered process; the performance is limited by the performance of the stable storage, however there is no upper limit to the number of processes which can simultaneously crash
- **storage-in-ram for homogeneous applications running on all nodes** [not yet fully implemented] - this setting may be used for transactional replication of a service. Since the copies of the same application run on each node and Paxos STM is used to ensure consistent state across all the replicas upon crash the process can retrieve and

apply the state of one of the other working processes. Since processes use snapshots of other processes in order to recover there is no need for creating ordinary checkpoints. Therefore calling the commit primitive does nothing. When all the processes function correctly the performance of Paxos STM in such settings is approximately the same as in case of crash-stop failure model. However the number of processes that can crash simultaneously have to be lower than the half of the number of the processes.

- **storage-in-ram for heterogeneous applications** [to be implemented] - processes save the history of exchanged messages in the volatile memory and from time to time they flash the logs to the stable storage. In the moments when all the processes are operational the logs can be shortened. The process may recover from the checkpoint made earlier or from the state of other process running the same application if it is operational. When all the processes are operational the performance of Paxos STM in such a setting should be similar to the performance in crash-stop model. The number of processes that can crash simultaneously depends on the model of distributed computing implemented in the applications.

Choosing suitable failure model requires setting the `FailureModel` property in JPaxos' configuration file:

```
FailureModel = CrashStop
```

for crash-stop failure model and:

```
FailureModel = FullStableStorage
```

for full stable storage failure model.

3.2 Building the library

The compilation of Paxos STM requires all of the listed in Requirements. libraries. Appropriate jar archives are placed in the lib directory.

The project can be build using Apache Ant. The follwing ant targets are available:

- [default] - compiles sources, generates documentation (see below) and creates jar archive,
- `build` - compiles source to the `bin` directory,
- `clean` - removes `bin`, `dist`, `javadoc` directories,
- `docs` - generates the documentation (sphinx in html format, javadocs); documenation to the project is placed in the `docs` directory whereas the *javadoc* documentation is located in the `javadoc` directory,
- `jar` - creates jar archive and places it in the `dist` directory,
- `zip` - creates the zip archive of the project and places it in the project's main directory,
- `zipnosource` - creates the zip archive of the project with jar archive but no sources and places it in the project's main directory.

In order to run a target execute the `ant <target>` command in the project's base directory. Calling `ant` without any arguments invokes the default target.

Application programming interface

There are four basic aspects of DSTM system's API:

- **'Transactions'_** - how to perform a transaction and how to change the control flow inside a transaction,
- Shared Objects - what can act as an object shared between the nodes and how it has to be defined,
- **'Incorporating Paxos STM framework into an application'_** - how to start the framework, utilize the recovery capabilities and make use of other features of Paxos STM,
- **'Paxos STM's behavior on error occurrence'_** - how does Paxos STM behave when the transactional code misbehaves, what happens when a/some replicas crash.

4.1 Atomic transactions

In Paxos STM framework transactions are defined as objects of classes derived from the abstract `Transaction` class. The `Transaction` class consists of the abstract `atomic()` method which has to be implemented by the programmer and is supposed the transaction's operation set.

The code inside the `atomic()` method can call several methods to alter the transaction's flow:

- `commit()` - ends the transaction earlier than the end of the `atomic()` method,
- `retry()` - restarts the transaction,
- `rollback()` - aborts the transaction,

```
class CustomTransaction extends Transaction {
  private MySharedObject sharedObject;

  public CustomTransaction(MySharedObject sharedObject) {
    this.sharedObject = sharedObject;
  }

  @Override
  protected void atomic() {
    sharedObject.foo();
    // commit();
    // retry();
    // rollback();
  }
}
```

The transaction is started when an instance of the class is created:

```
new CustomTransaction(sharedArray);
```

The `Transaction` class contains also a parametrized constructor which allows to delay the execution of a newly created transaction. In such a case the transaction needs to be started manually by the programmer by calling the `execute()` method on the instance of the class.

It is most convenient to use Java's anonymous classes to define transactions as shown on the example below:

```
final MySharedObject sharedObject = new MySharedObject();

new Transaction() {
  protected void atomic() {
    sharedObject.foo();
    // commit();
    // retry();
    // rollback();
  }
};
```

Notice the use of the `final` keyword in the declaration of shared objects. This is required by the Java semantics in order to pass variables to the anonymous class' methods.

4.2 Shared objects

There are three kinds of objects that can be shared between the nodes:

instances of classes marked with the `@TransactionObject` annotation

Marking a class definition with the `@TransactionObject` annotation advices Paxos STM framework to alter the class definition when loading it to JVM. This allow automatic tracking of changes performed to the objects in transactions.

Instances of classes marked with the `@TransactionObject` annotation are created normally with the `new` operator:

```
@TransactionObject
class MySharedObject {
  public int value;

  public void foo() {
  ...
  }
  ...
}

MySharedObject sharedObject = new MySharedObject();
```

Objects of classes marked with the `@TransactionObject` annotation can be accessed inside the transaction directly, that is as ordinary Java objects in native Java code:

```
final MySharedObject sharedObject = new MySharedObject();

new Transaction() {
  protected void atomic() {
    ...
    sharedObject.value = 5;
    ...
    sharedObject.foo();
    ...
  }
};
```

instances of `ObjectWrapper` class

The `@TransactionObject` annotation can be added to the definition of the class only when the programmer has the control over it. It cannot be done with system classes or classes from third-party libraries. It is due the way classes are loaded to JVM in Paxos STM and the Java license which forbids performing modification to the system classes (for details see [KKW10]). Therefore, in order to allow using objects of system classes or classes from external libraries as objects shared between the nodes, the solution based on `ObjectWrappers` is provided. Objects of any class can become shared objects as long as they are *wrapped* inside the `ObjectWrapper`:

```
Vector<String> vector = new Vector<String>();
vector.add("foo");
vector.add("bar");
ObjectWrapper<Vector<String>> sharedVector = new
  ObjectWrapper<Vector<String>>(vector);
```

Objects wrapped inside the `ObjectWrappers` has to be accessed by calling the `get()` method on the `ObjectWrapper` object:

```
...
final ObjectWrapper<Vector<String>> sharedVector = new
  ObjectWrapper<Vector<String>>(vector);

new Transaction() {
  protected void atomic() {
    sharedVector.get().add("foo and bar");
    System.out.println(sharedVector.get().size());
    System.out.println(sharedVector.get().get(0));
  }
};
```

instances of `ArrayWrapper` class

Arrays cannot be treated as ordinary objects therefore they have to be handled differently. Similarly to objects of system classes or classes from external libraries, arrays have to be stored inside wrappers in order to use them inside transactions. The array-specific wrapper is called `ArrayWrapper`:

```
int[] array = new int[] {2, 3, 5};
ArrayWrapper<Integer> sharedArray = new
  ArrayWrapper<Integer>(array);
```

Array elements have to be accessed by calling the `T get(int index)` and `void set(int index, T value)` methods. The array stored inside the `ArrayWrapper` can be temporarily released by calling the `release()` method so elements of the array can be modified directly. The array can be again locked inside the `ArrayWrapper` object by calling the `hold()` method:

```
int[] array = new int[] {2, 3, 5};
final ArrayWrapper<Integer> sharedArray = new
  ArrayWrapper<Integer>(array);

new Transaction() {
  protected void atomic() {
    int firstValue = sharedArray.get(0);
    sharedArray.set(1, firstValue);
    int []a = sharedArray.release();
    System.out.println("[" + a[0] + ", " + a[1] + ", " + a[2] + "]");
    a[2] = 7;
    sharedArray.hold();
    sharedArray.set(2, 77);
  }
};
```

4.3 Writing an application program

Using Paxos STM in an application is not only related to creating shared objects and performing transactions. Firstly, the Paxos STM framework has to be overtly started by the application. Secondly, the application has to implement a set of interfaces in order to profit from the Paxos STM recovery capabilities. Thirdly, the programmer can use the so called *Shared Object Registry (SOR)* to associate labels (of the type of `java.util.String`) with shared object so they can be easily accessed.

starting the Paxos STM framework

Starting Paxos STM requires invoking the `start()` method on the instance of the `Paxos STM` class which is implemented using the singleton pattern:

```
PaxosSTM.getInstance().start();
```

providing recovery capabilities for the whole application

In order to provide recovery capabilities for the whole application the programmer is required to do the following:

- select crucial data that constitute the application's state and provide methods for storing and retrieving them from the stable storage,
- synchronize the process of storing/retrieving the data from the stable storage with analogue processes in the Paxos STM framework.

To illustrate how it should be performed let us consider the following example. The whole program is confined in `ExampleClass`. The processing of the application is reduced to incrementing a value in an infinite loop; to shorten the example it is not even performed transactionally. After each incrementation the program saves its state to the stable storage.

```
public class ExampleClass implements CommitListener, RecoveryListener
{
  private Storage storage;
  private Object recoveryLock = new Object();
  private boolean ready = false;

  private int counter;

  public ExampleClass()
  {
    PaxosSTM.getInstance().addCommitListener(this);
    PaxosSTM.getInstance().addRecoveryListener(this);
    this.storage = PaxosSTM.getInstance().getStorage();
  }

  @Override
  public void onCommit(Object commitData)
  {
    int commitNumber = (Integer) commitData;
    int logVersion = commitNumber % 2;

    try
    {
      storage.log("Counter_" + logVersion, counter);
    }
    catch (StorageException e)
    {
      e.printStackTrace();
    }
  }

  @Override
  public void recoverFromCommit(Object commitData)
  {
    int commitNumber = (Integer) commitData;
    int logVersion = commitNumber % 2;

    try
    {
      counter = (Integer) storage.retrieve("Counter_" + logVersion);
    }
    catch (StorageException e)
    {
      counter = commitNumber;
    }
  }

  @Override
  public void recoveryFinished()
  {
    synchronized (recoveryLock)
    {
      ready = true;
      recoveryLock.notifyAll();
    }
  }

  public void start()
  {
    try
    {
      PaxosSTM.getInstance().start();
    }
    catch (StorageException e)
    {
      e.printStackTrace();
      System.exit(1);
    }

    synchronized (recoveryLock)
    {
      while (!ready)
      {
        try
        {
          recoveryLock.wait();
        }
        catch (InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }

    while (true)
    {
      // do some processing
      counter++;

      PaxosSTM.getInstance().commit(counter);
    }
  }

  public static void main(String[] args)
  {
    ExampleClass instance = new ExampleClass();
    instance.start();
  }
}
```

Paxos STM framework provides a set of interfaces which have to be implemented so the process of creating checkpoints/recovering could be coordinated throughout the whole application:

- `CommitListener` - defines the method `onCommit()` which consists of the code necessary to save to the stable storage the application's state
- `RecoveryListener` - defines two methods: `recoverFromCommit()` which consists of code necessary to restore from stable storage application's state and `recoveryFinished()` which signalizes that the process of recovering is finished and the application can resume its duties.

The programmer can use the Storage interface provided by Paxos STM in order to save necessary data:

```
Storage storage = PaxosSTM.getInstance().getStorage();

MyClass obj = new MyClass();
storage.log("label", obj);
MyClass obj2 = (MyClass) storage.retrieve("label");

// obj.equals(obj2) == true
```

The application can trigger the process of creating a checkpoint by calling the `commit()` method:

```
Object commitData; // application snapshot or counter variable
...
PaxosSTM.getInstance().commit(commitData);
```

The argument of the `commit()` method should be the number of checkpoint to be created. This guarantees that the consecutive commits will be numbered consistently also in case of a crash and recovery. The interface providing checkpoint capabilities is designed in a way so the checkpoints can be also created by passing a snapshot of the application to the Paxos STM, thus the `Object` as the type of the `commitData`. In order to guarantee consistency at all times at least the last two checkpoints have to be preserved on the stable storage.

Notice that the `ExampleClass` in its constructor adds itself as Paxos STM `Commit-` and `RecoveryListener` thus allowing the Paxos STM to coordinate the process of creating checkpoints and recovering. The simple synchronization present in the example above is necessary so the application do not compute in the recovery phase.

adding/removing a shared object to/from Shared Object Registry

Operations on Shared Object Registry (SOR) are performed by appropriate methods on the instance of the `PaxosSTM` class. In the example below a process on one node (`node1`) adds an object to the registry by the label of `MyObjectA`. Then any other process can retrieve the reference to this object by calling the registry and passing the appropriate label (`MyObjectA`) as the argument:

```
// on node1
MySharedClass sharedObjectA = new MySharedClass();
PaxosSTM.getInstance().addToRegistry("MyObjectA", sharedObjectA);

// any process
MySharedClass sharedObjectA = (MySharedClass)
  PaxosSTM.getInstance().getFromRegistry("MyObjectA");
```

A reference to an object can be removed from the registry:

```
// any process
MySharedClass sharedObjectA =
  PaxosSTM.getInstance().removeFromRegistry("MyObjectA");
```

4.4 Handling errors and failures

handling exception in transactional code

The programmer should secure the transactional code so there are no unhandled exceptions. However, if by any chance an unhandled exception is thrown inside the `atomic()` method it is caught by the Paxos

STM and the whole transaction is rolled back.

crash of a/some replicas

JPaxos, on top of which Paxos STM is built, to be able to operate correctly requires at least one more of the half of all the nodes to be up and running. So if there are three replicas all together, if one crashes, Paxos STM can still function normally. On the other hand if two crash simultaneously, the other replica withholds performing commit attempts on any currently executed transaction. Operating on Shared Object Registry is also suspended. Paxos STM resumes all its duties once at least one of the crashed replicas recovers.

Executing an application program

Let us assume there is an application samplepackage.App that employs Paxos STM. The invocation of the program should look as below:

```
$ java -cp "JPaxos.jar:PaxosSTM.jar:javassist.jar:objenesis-1.2.jar:
  <path to the App jar archive or directory with compiled classes>"
    soa.paxosstm.dstm.Main
      samplepackage.App <replica number> [samplepackage.App arguments]
```

In order to disable showing log information on the console the following system property should be defined (logging_none.properties file is in the project's main directory):

```
-Djava.util.logging.config.file=logging_none.properties
```

In the `utils/unix` and `utils/windows` directories there is a bunch of useful scripts (cygwin scripts for the windows platform), e.g. one that helps to start an application employing Paxos STM:

```
$ ./run.sh samplepackage.App <replica number> [samplepackage.App arguments]
```

Other scripts are provided only for testing purposes.

CHAPTER 6

Bibliography

Bibliography

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In Proceedings of PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August 1995.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[M06] Sergio Mena de la Cruz. Protocol composition frameworks and modular group communication. PhD thesis, EPFL, EPFL, 2006.

[KKW09] Tadeusz Kobus & Maciej Kokocinski & Pawel T. Wojciechowski (2009): Distributed Software Transactional Memory in Crash-recovery Failure Model. Technical Report TR-ITSOA-OB2-1-PR-09-7, Instytut Informatyki, Politechnika Poznanska.

[KKW10] Tadeusz Kobus & Maciej Kokocinski & Pawel T. Wojciechowski (2010): Paxos STM. Technical Report TR-ITSOA-OB2-1-PR-10-4, Instytut Informatyki, Politechnika Poznanska.

[JPaxosWeb] http://jpaxos.org

[C09] M. Couceiro. Cache coherence in distributed and replicated transactional memory systems, 2009.